

# GSOC Raspberry Pi - Rock, Paper, Scissors

Last updated June 17, 2020

## Congratulations on setting up the Raspberry Pi !!!

Looking for another coding challenge? This project is fun and shouldn't take more than 60 minutes for you to complete. We will be coding a CLIENT / SERVER game of rock, paper and scissors!

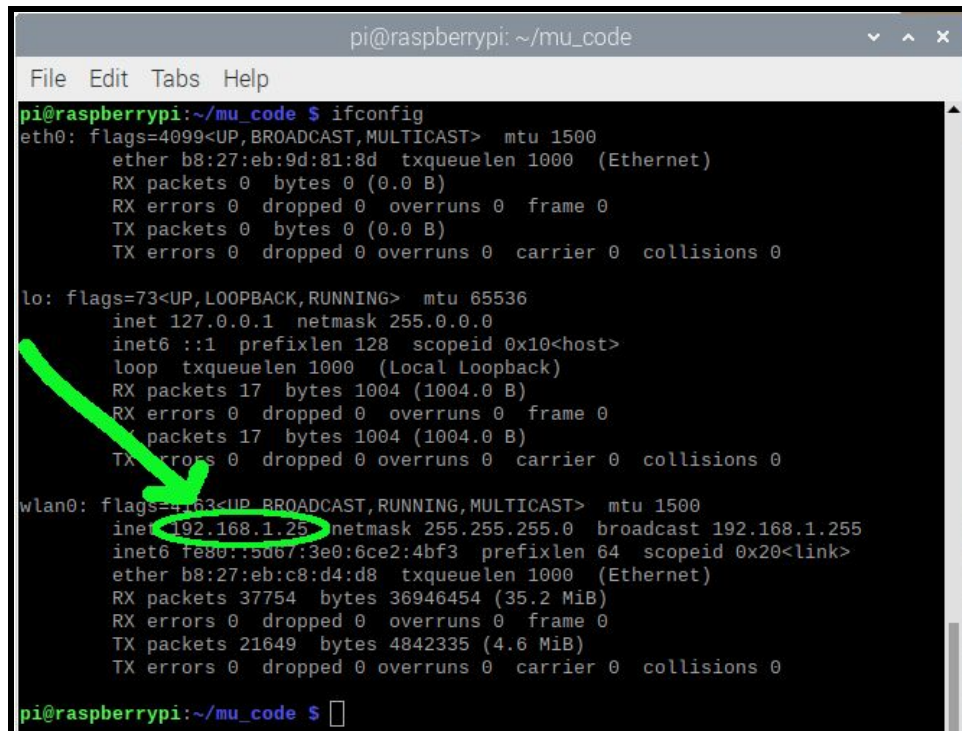
### Prerequisites:

- At least TWO (more == more fun!) Raspberry Pi's
- A WiFi network that all of the Raspberry Pi's will connect to
- Completion the "Box setup Instructions" for all of the Raspberry Pi's
- Complete the "WiFi connectivity Instructions" for all of the Raspberry Pi's
- Connection of all of the Raspberry Pi's to the same WiFi network

## **STEP #1 Setup the SERVER**

Select one of the Raspberry Pi's as the "server". This Raspberry Pi will be receiving requests from each of the other Raspberry Pi's.

Open a terminal window (click on the black icon in the upper left corner of the screen) and type the command `ifconfig` at the terminal prompt as shown below:

A screenshot of a terminal window titled 'pi@raspberrypi: ~/mu\_code'. The window shows the output of the 'ifconfig' command. A green arrow points to the IP address '192.168.1.25' in the 'wlan0' section, which is also circled in green. The terminal output shows details for the 'eth0', 'lo', and 'wlan0' interfaces, including their flags, MTU, and various statistics.

```
pi@raspberrypi: ~/mu_code
File Edit Tabs Help
pi@raspberrypi:~/mu_code $ ifconfig
eth0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    ether b8:27:eb:9d:81:8d txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 17 bytes 1004 (1004.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 17 bytes 1004 (1004.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.1.25 netmask 255.255.255.0 broadcast 192.168.1.255
    inet6 fe80::5067:3e0:6ce2:4bf3 prefixlen 64 scopeid 0x20<link>
    ether b8:27:eb:c8:d4:d8 txqueuelen 1000 (Ethernet)
    RX packets 37754 bytes 36946454 (35.2 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 21649 bytes 4842335 (4.6 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

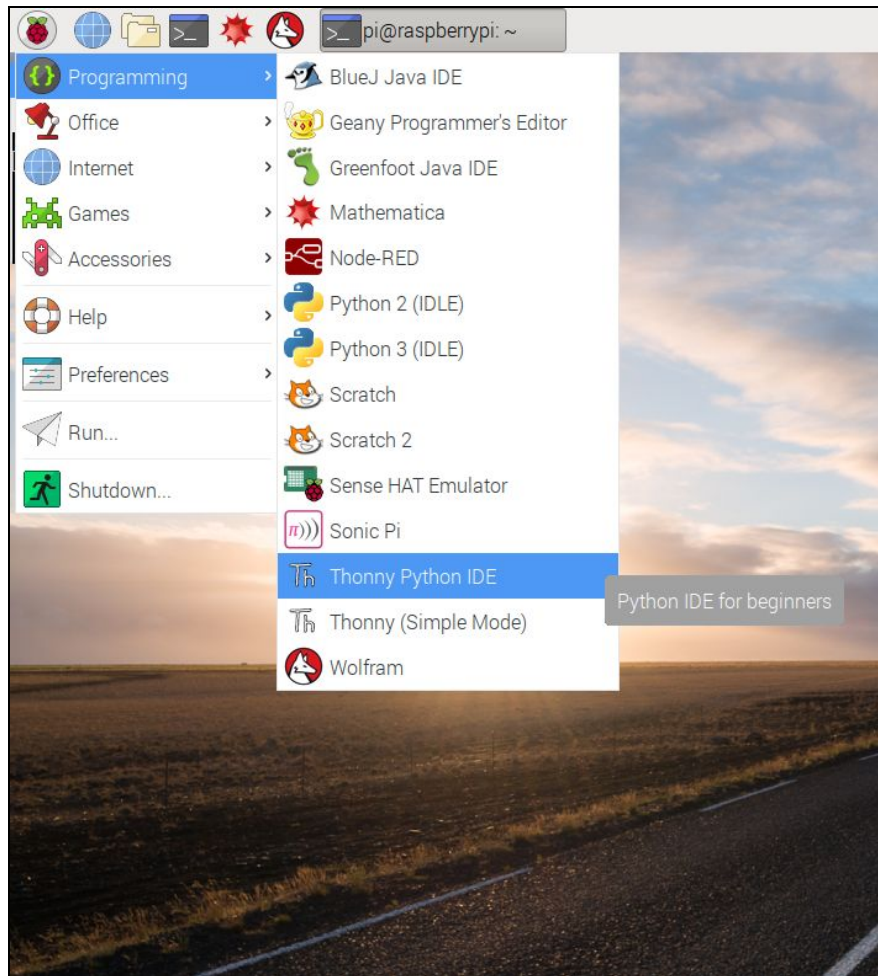
pi@raspberrypi:~/mu_code $
```

Note the values circled above in the "wlan0:" section of the terminal output. This number (192.168.1.25) is the IP ADDRESS of the Raspberry Pi server on the local wifi network. NOTE: Your Raspberry Pi's IP address will almost certainly be DIFFERENT than this one!

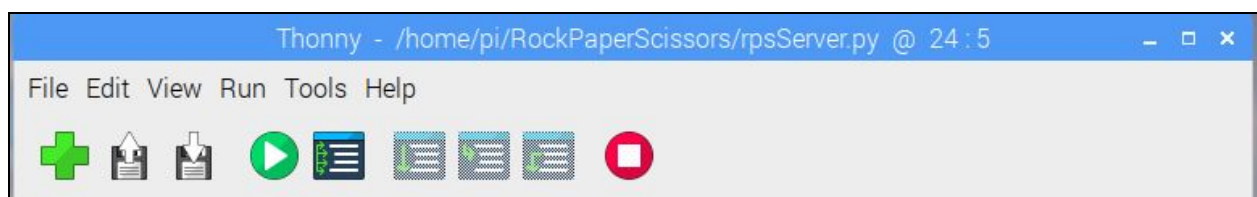
Be sure that you carefully record this IP address, as you will need to enter it into both the server's Python code and the "clients" (i.e. the other Raspberry Pi's) javascript code.

## **STEP #2 Code the SERVER (getting started)**

Click on the Raspberry in the upper left corner of the screen to display the system menu. Pick Programming and then "Thonny Python IDE" menu items as shown in the image below.



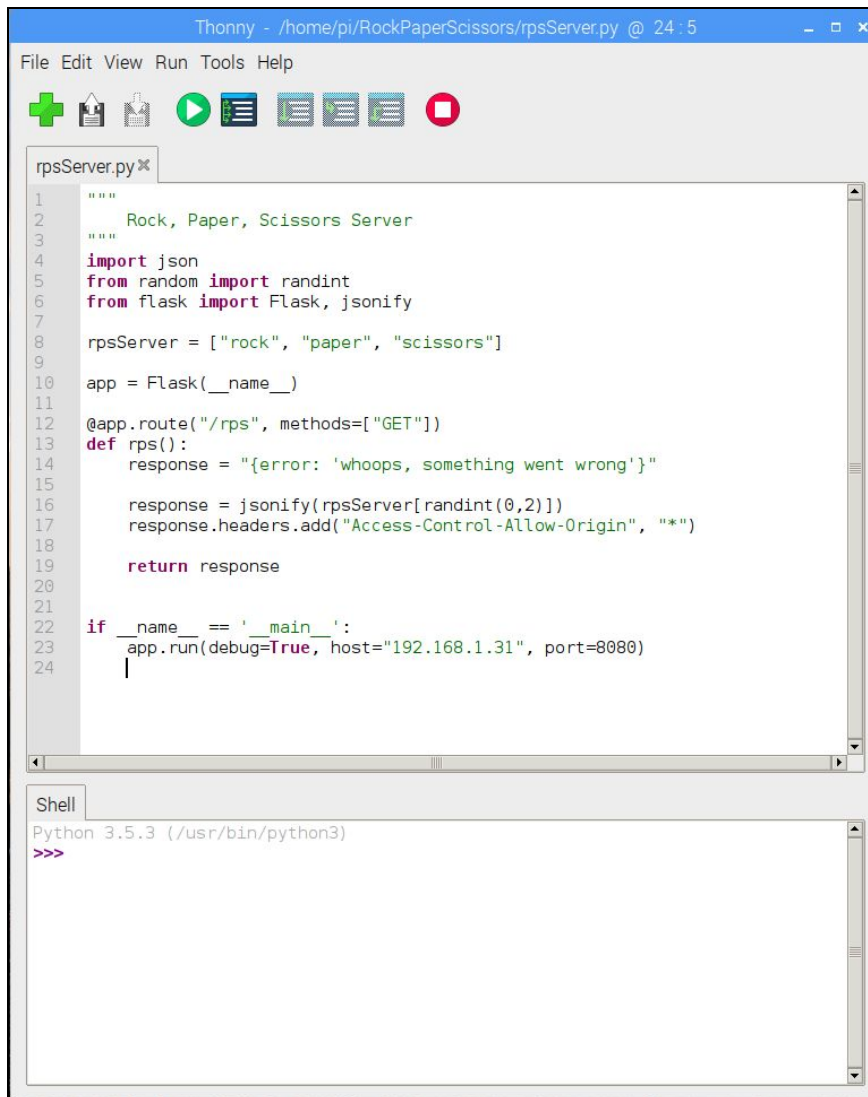
Click on the Thonny editor's NEW FILE button (green plus sign shown on the left below):



Click on Thonny's SAVE button (third tool from left - the diskette icon with DOWN arrow) and save your program file as "rpsServerFL" **where F is the first character of your first name and L is the first character of your last name.** The name rpsServerFL.py will appear in the editor window's tab.

### **STEP #3 Code the SERVER (copying the code)**

Carefully copy the program displayed on the next page into the Thonny's editor window. A line by line explanation of the program is provided beneath the image below.



```
Thonny - /home/pi/RockPaperScissors/rpsServer.py @ 24:5
File Edit View Run Tools Help

rpsServer.py
1  """
2      Rock, Paper, Scissors Server
3  """
4  import json
5  from random import randint
6  from flask import Flask, jsonify
7
8  rpsServer = ["rock", "paper", "scissors"]
9
10 app = Flask(__name__)
11
12 @app.route("/rps", methods=["GET"])
13 def rps():
14     response = '{"error: 'whoops, something went wrong'}'
15
16     response = jsonify(rpsServer[randint(0,2)])
17     response.headers.add("Access-Control-Allow-Origin", "*")
18
19     return response
20
21
22 if __name__ == '__main__':
23     app.run(debug=True, host="192.168.1.31", port=8080)
24
Shell
Python 3.5.3 (/usr/bin/python3)
>>>
```

Lines 1 - 3 represent a “comment”, meaning the lines do not execute or “run” as part of the code. These lines included merely to help us remember what this program is for. You do not need to include these three lines if you don’t want to. Do note that line 1 and line 3 contain three quote characters, and line 2 is indented (hit the tab key before typing the text to indent). Comments in Python are either implemented this way (most typically way for multiple line comments) or start with a # character within a single line.

Python allows you to leverage the work of others (or yourself) by including modules, or other pieces of code, within your program. Almost every python program starts out this way! In lines 4 - 6 we import the modules that our program will use.

On line 8 we declare a new variable called rpsServer, representing the choices (in this case 3) that the server can respond back to the client computers. The rpsServer variable is a python

LIST containing 3 list elements, “rock”, “paper” and “scissors”. Our server will randomly return one of these three choices.

On line 10 we create a new “app” object using one of our imported modules called Flask. Line 12 is the starting point of our “rps” function which we declare on line 13. Line 12 indicates that our server’s URL “endpoint” to call the rps function will be “rps”. We’ll talk more about the server end point when we go over our client javascript code.

Lines 13 - 19 represent our rps function, or the function that is called when our clients connect to this server program. Note that lines 14 - 19 are indented beneath the function declaration on line 13, which is Python’s convention. On line 14 we declare a new variable called “response” representative of the information the server will send back to the client. We initially set the variable “response” to an error message.

On line 16 we do two things: (1) randomly select an item from our previously declared list rpsServer and (2) convert the selected list item to a data format known as JSON using the flask module’s “jsonify” (JSON - ify this stuff) function. The function randint(0,2) randomly selects a number from 0 through (and including) 2, meaning it will randomly select one of the list variable rpsServer list items: 0 == “rock”, 1 == “paper” or 2 == “scissors”.

We need to add line 17 to complete the communications between our server and clients.

On line 19 we return the value of the response variable back to our client.

On line 22 we signify where our program starts, and on line 23 we call the “run” method of the app object we created on line 10.

***Important note: You will need to change the “192.168.1.25” on line 23 to your server’s IP address, as we determined in STEP #1. Be sure to replace “192.168.1.25” with your server’s IP address!***

Here’s the entire rpsServer.py code listing as a reference:

```
"""
    Rock, Paper, Scissors Server
"""
import json
from random import randint
from flask import Flask, jsonify

rpsServer = ["rock", "paper", "scissors"]

app = Flask(__name__)

@app.route("/rps", methods=["GET"])
def rps():
    response = "{error: 'whoops, something went wrong'}"

    response = jsonify(rpsServer[randint(0,2)])
```

```

response.headers.add("Access-Control-Allow-Origin", "*")

return response

if __name__ == '__main__':
    app.run(debug=True, host="192.168.1.25", port=8080)

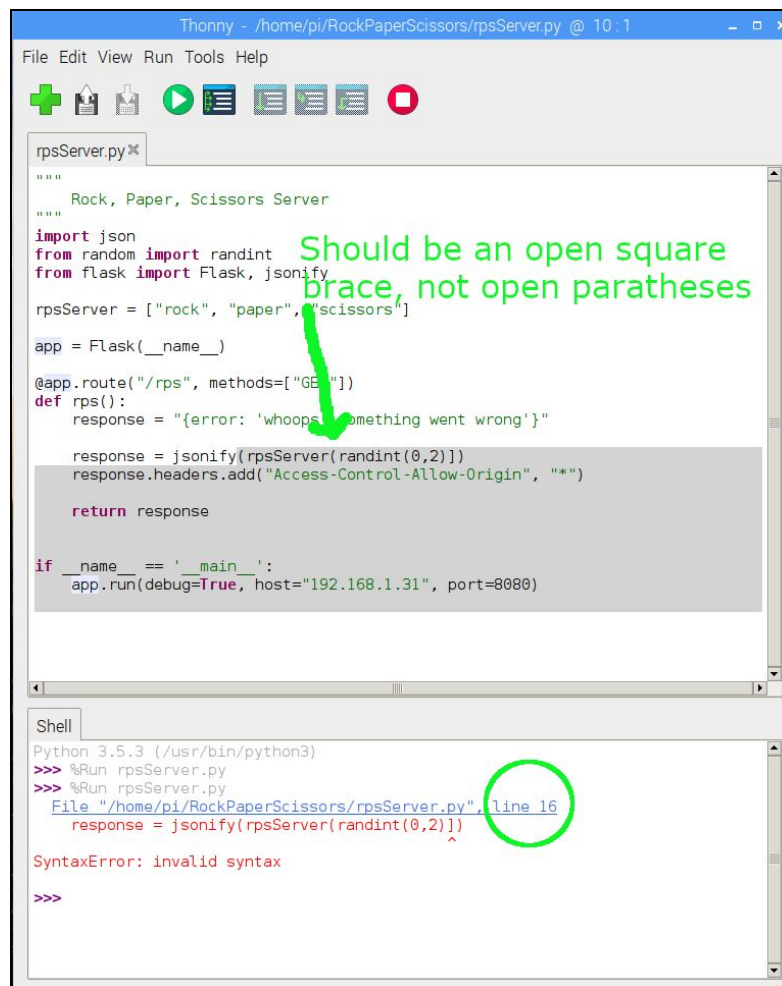
```

## **STEP #4 Start the Server**

Click on the GREEN / WHITE ARROW toolbar icon, which is Thonny's "run" button.

One of two things will happen when you click the RUN button. Most likely, you will have made an error copying the code and the Thonny editor will display the error within the bottom window of the editor. Don't worry about making an error, this is part of learning how to code. It is nearly impossible to copy a program of this size without making any errors, and the errors you make will help you better understand the code.

If you have an error, it will appear as the example shown below. Note that the error messaging normally indicates a LINE NUMBER which can help you identify where you may have made a mistake in copying the code. In the example below, we inadvertently typed an open parentheses instead of typing an opening square brace.






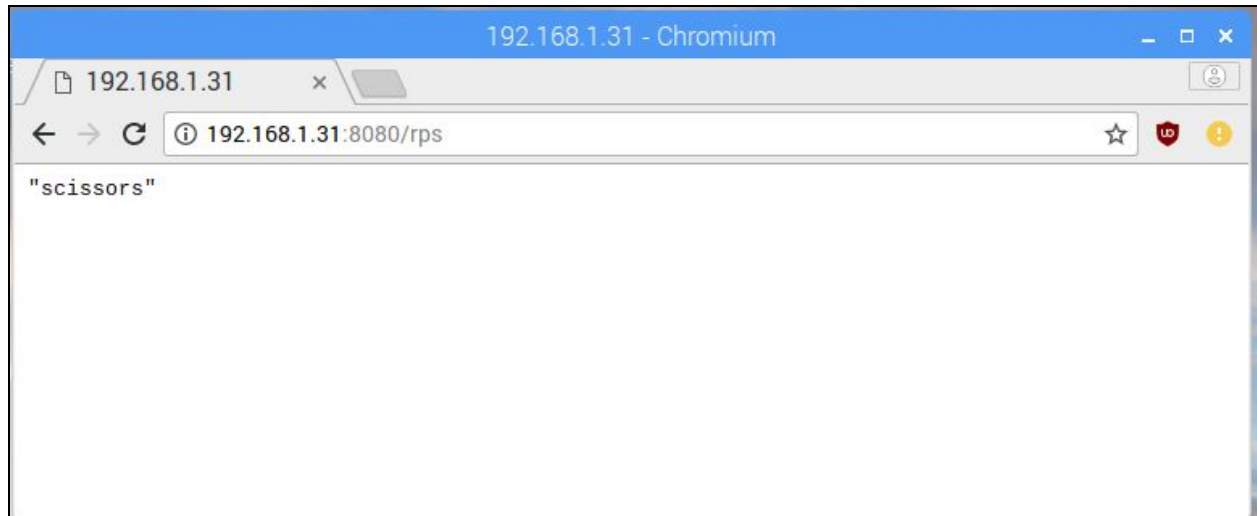
Click the RED STOP BUTTON to be sure your code is not running, then correct the error in the Thonny editor and try to re-run your code. Don't forget to SAVE your file as you make changes; you can use the third toolbar tool (diskette with the down arrow) to save your file.

If you don't see any error messages (and instead, a message that says %Run rpserverFL.py), then you can presume the server is running.

Since there is no messaging, it's a good idea to confirm that the server is running. We can easily do this using the Raspberry Pi's Chromium web browser. The web browser can be

opened by clicking on the Raspberry Pi toolbar's blue "globe" icon . Enter the following URL as shown in the picture below, **substituting 192.168.1.31 with YOUR SERVER'S IP ADDRESS**:

```
http://192.168.1.31:8080/rps
```




Note that the server returned "scissors". If you click on the circular REFRESH button just to the left of the URL you typed in, you may receive a different message (i.e. either "rock", "paper" or "scissors") reflective of the randomization within our server code at line 16.

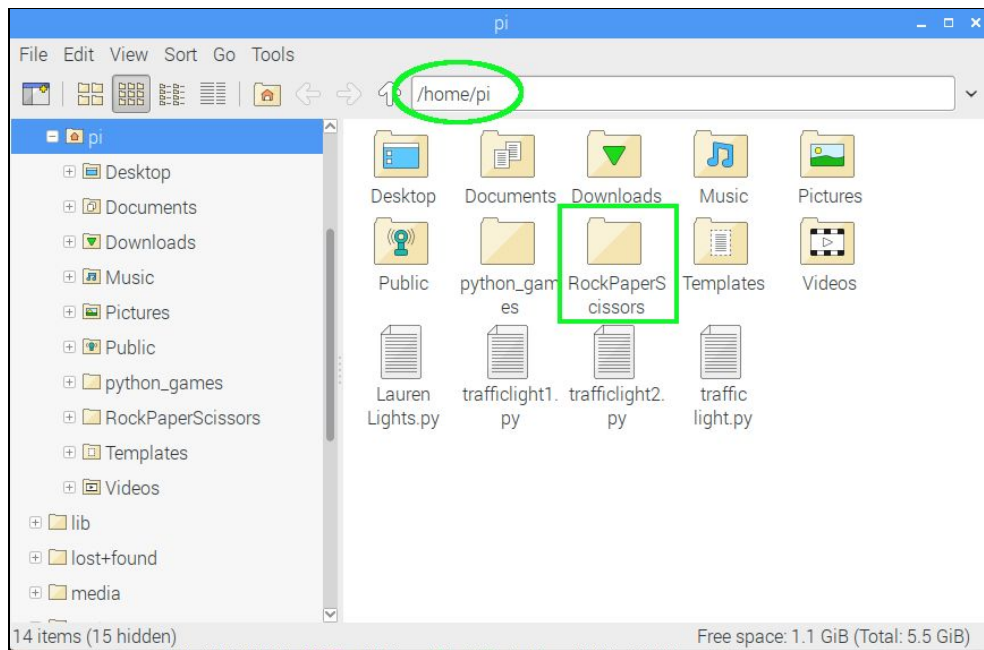
Now let's build a more interesting "client side" on our other Raspberry Pi's and start playing rock, paper, scissors against our now operational server!

## **STEP #5 Setup the Client(s) Raspberry Pi computers**


Move on to any of the "client" Raspberry Pi computers and open the File Manager program

using the "file folders" icon on the toolbar .

Navigate to the folder /home/pi/ and see if the folder RockPaperScissors exists within the /home/pi folder, as shown in the picture below.



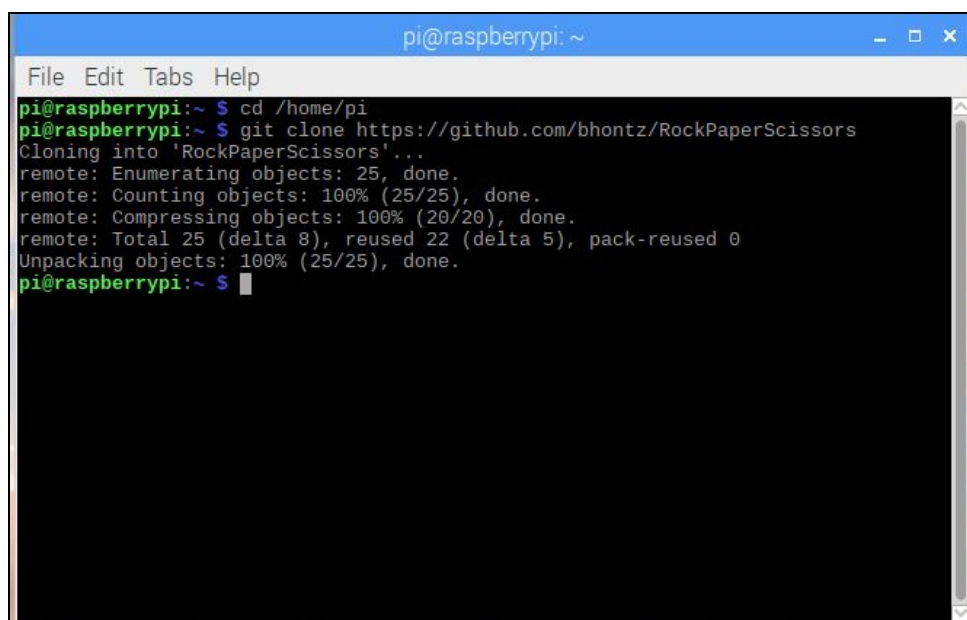
If the RockPaperScissors folder already exists, **skip ahead to STEP #6**. If not, then click on the

“terminal” toolbar icon  and enter the following two commands, using the ENTER key to submit each command.

```
cd /home/pi
```

```
https://github.com/bhontz/RockPaperScissors
```

Messages will appear within the terminal window similar to the picture below.

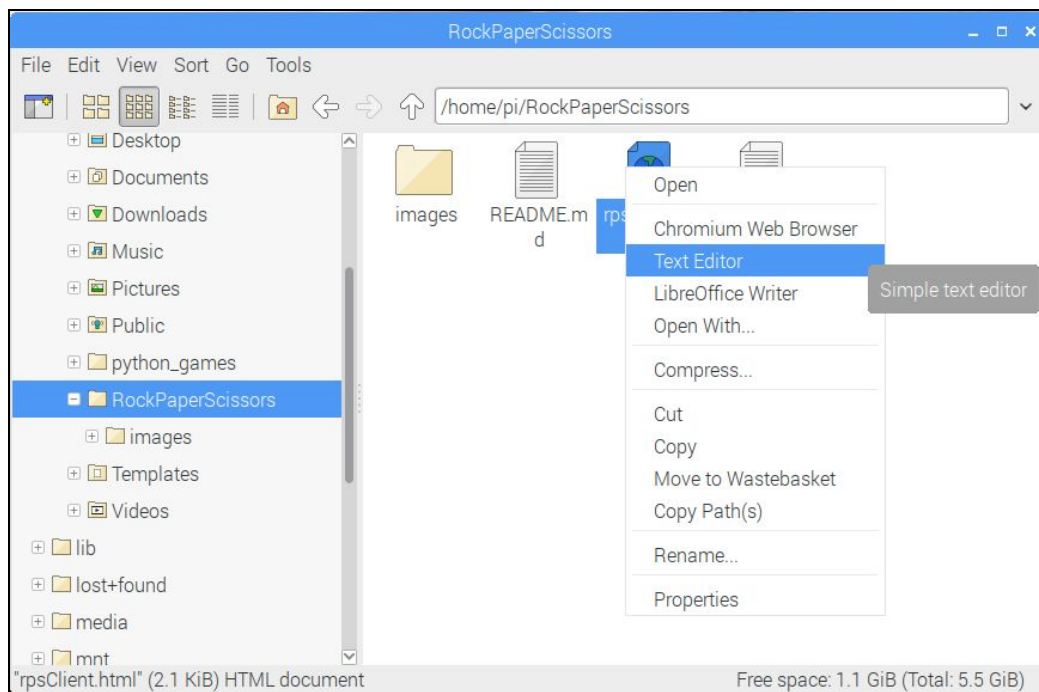


```
pi@raspberrypi:~  
File Edit Tabs Help  
pi@raspberrypi:~ $ cd /home/pi  
pi@raspberrypi:~ $ git clone https://github.com/bhontz/RockPaperScissors  
Cloning into 'RockPaperScissors'...  
remote: Enumerating objects: 25, done.  
remote: Counting objects: 100% (25/25), done.  
remote: Compressing objects: 100% (20/20), done.  
remote: Total 25 (delta 8), reused 22 (delta 5), pack-reused 0  
Unpacking objects: 100% (25/25), done.  
pi@raspberrypi:~ $
```

Close the terminal window by clicking on the X in the upper left corner of the window. Before moving on to STEP #6, return to the beginning of the instructions for STEP #5 to verify that the RockPaperScissors folder is now available within the /home/pi directory.

## **STEP #6 Edit file rpsClient.html**

Return to the File Manager program, and click on the RockPaperScissors folder in order to open it as shown below. Right click on the file rpsClient.html found within the RockPaperScissors folder and select the menu item Text Editor to open and edit the rpsClient.html file.



The rpsClient.html will appear with the Text Editor as shown in the picture below. While we will go over this file line by line, but for now, simply change line 23:

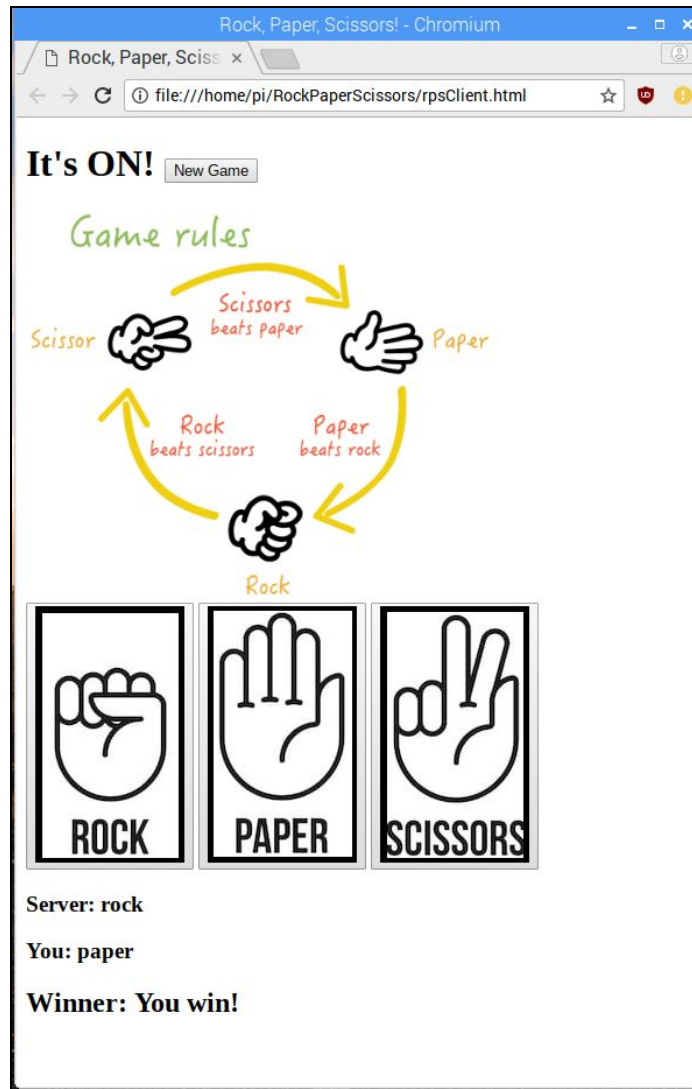
```
myRequest.open( 'GET', 'http://XXX.XXX.XXXX:8080/rps' );
```

... by replacing the XXX.XXX.XXXX with the server's IP address (as obtained in STEP #1)

Save the file using the Text Editor's File Save menu commands, and then close the Text Editor using the Text Editor's File Quit command.

You can now use the File Manager to double click on the revised rpsClient.html file, which will open the Chromium Web Browser and appear as in the next picture.





Click on the New Game button to begin **EACH NEW GAME**. Now click on either the Rock, Paper or Scissors buttons to make your selection, and view the server's response, your selection, and the "winner" in the fields immediately beneath these three buttons.

See how many Raspberry Pi clients you can configure to play against the server at the same time! You will need to set up each of the Raspberry Pi clients in the exact same way as we have in **STEPS #5 and #6**.

### **So how does the rpsClient.html file work?**

The rpsClient.html file contains HTML and Javascript code. A web browser understands how to interpret HTML and Javascript and render the results within the web browser's application window. The next two screenshots illustrate the rpsClient.html code along with line numbers.

```

1 <!DOCTYPE html>
2 <html>
3
4 <head>
5     <meta charset="utf-8">
6     <title>Rock, Paper, Scissors!</title>
7 </head>
8
9 <script>
10     var yourPick = "nothing";
11
12     function myPick(weapon) {
13         myRequest.send();
14         document.getElementById('your-selection').innerHTML = weapon;
15         yourPick = weapon;
16     }
17
18     var myRequest = new XMLHttpRequest();
19     //
20     // * * * HEY GIRLS, REPLACE XXX.XXX.XXXX IN THE LINE BELOW * * *
21     // * * * WITH YOUR SERVER'S IP ADDRESS ! * * *
22     //
23     myRequest.open('GET', 'http://192.168.1.31:8080/rps');
24     myRequest.onreadystatechange = function () {
25         if (myRequest.readyState === 4) {
26             var serverPick = myRequest.responseText;
27             serverPick = serverPick.split('').join('').trim();
28             document.getElementById('server-selection').innerHTML = serverPick;
29
30             if (serverPick == yourPick) {
31                 document.getElementById("winnerIs").innerHTML = "It's a TIE!";
32
33             } else if (serverPick == "scissors" && yourPick == "paper") {
34                 document.getElementById("winnerIs").innerHTML = "Server wins!";
35
36             } else if (serverPick == "paper" && yourPick == "rock") {
37                 document.getElementById("winnerIs").innerHTML = "Server wins!";
38
39             } else if (serverPick == "rock" && yourPick == "scissors") {
40                 document.getElementById("winnerIs").innerHTML = "Server wins!";
41
42             } else {
43                 document.getElementById("winnerIs").innerHTML = "You win!";
44             }
45         }
46     }
47 </script>
48

```

HTML files consist of “tags” which are statements enclosed within < > “pointy brackets”. Most HTML tags are used within pairs, with the starting tag representing the beginning and the ending tag the end of the tagged section’s domain. For example, all HTML files need to start with an <html> tag and end with a closing </html> tag. Note how the backslash designates the closing tag.

The <head></head> section within lines 4 - 7 define the character set used within the file (line 5) and the title of the HTML file, which is displayed within the browser’s tab for this file (refer to the browser tab shown in the last picture of STEP #6 to confirm). Feel free to change the text between the <title></title> tags and see how it impacts the browser tab after you refresh the page.

The `<script></script>` section within lines 9 - 47 contains both the game logic (lines 30 - 44) and the means of obtaining the server's response (lines 18 - 25).

The function declared on lines 12 - 16 is called each time one of our three rock, paper or scissors buttons is pressed. We'll show where that happens within the second screenshot. When we press one of the three buttons, we execute line 13, which sends an HTML 'GET' command to our server.

On line 18 we declare a new server request object that we name "myRequest". On line 19 we setup the GET request using the server's IP address, and then we declare a function that "waits" for the request. This function comprises lines 24 - 46.

If this waiting function finds that the server returns a "ready state" equal to the value of 4 (which means the server request finished and a response is available) we then obtain the server's response in our newly declared variable *serverPick*. The response from the server is wrapped in double quotes, which we remove on line 27. On line 28, we display the value we obtained from the server in the section of the page that has the identifier "server-selection".

Lines 33 through 44 comprise the "logic" of our game. Here we compare the server's response (*serverPick*) to our own selection, which is set to the variable called *yourPick*. The variable *yourPick* is set within the function *myPick* (lines 12 - 16), which is called when we press one of the three buttons on the page.

If *serverPick* is identical to *yourPick*, we display "It's a TIE!" in the section of the page with the identifier "winnerIs". Otherwise (else if) we check the conditions specific to the server winning and display "Server wins!" within the "winnerIs" section of the page. If none of the "server winning" conditions exist (lines 33 - 40) then we assume *yourPick* was the winner, and we display "You win!" within the "winnerIs" section of the page. Note that we could have written the logic to check for the conditions in which *yourPick* would have won, leaving the default "else" condition to assume the server won.

The line numbered screenshot below shows the remainder of the *rpsClient.html* file (lines 47 overlap between the two screenshots).

```
47 </script>
48
49 <body>
50   <h1>It's ON! <button onclick="location.reload();">New Game</button></h1>
51   <div></div>
52   <button onclick="myPick('rock');"></button>
53   <button onclick="myPick('paper');"></button>
54   <button onclick="myPick('scissors');"></button>
55   <br>
56   <h3>Server: <span id="server-selection"></span></h3>
57   <h3>You: <span id="your-selection"></span></h3>
58   <h2>Winner: <span id="winnerIs"></span></h2>
59 </body>
60
61 </html>
62
```

The `<body></body>` section within lines 49 - 59 represent what is actually displayed on the web browser's screen. Compare this section to the last picture within STEP #6 and you'll see how the page layout works. Line 50, or the top of our web page, contains both the "It's ON!" text and the button that starts a New Game. When clicked, this button simply reloads the page, which reruns the `<script></script>` section of the page. The `<h1></h1>` tags on line 50 set a "heading style" or large font, for this section of the page.

On line 51 we add an image to the page. The image is found within the subfolder "images" (beneath the RockPaperScissors folder) and is called rock-scissors-paper-game-rules.png. You can use the Raspberry Pi's File Manager to find and view this file if you'd like to.

Note that the `<h1></h1>` as well as the `<div></div>` tags start a new line immediate after their closing tag. Alternatively, the next three lines (52 - 54) of our code, which represent our three buttons, appear on the same web page line; in other words, all three buttons are arranged horizontally.

Our three button tags contain a "onclick" reference to the aforementioned function "myPick()" which is declared on lines 12 - 16. We pass an argument to the myPick function which is specific to the button's image; for example, we pass the argument 'rock' for the button containing the rock image. The images (img tag) are declared inside the button tags, and like the rock-scissors-paper-game-rules.png, the image files associated with the images (e.g. rock.jpg) are found within the images subfolder of the RockPaperScissors folder.

The tag `<br />` on line 55 is called "break row" and simply adds a newline, or vertical spacing, to our page. Note that this tag contains the backslash closing character within the tag, allowing us to use a single tag.

The next three lines, 53 - 58 contain a label and `<span></span>` area where our script variables are displayed. The span tags each have an "id", and the id links this area of the webpage to the variables within our script.

Line 61 contains our closing `</html>` tag and signifies the end of our HTML file.

## **So why is this cool ? (i.e. what did we learn)**

This project is a simple introduction to Client / Server network computing. Within the Client / Server model, the client always initiates a request for information from the server. The server returns a status code, and (usually) data that the client can interpret and deal with. Client / Server computing is extremely common and useful. Consider a very complicated process, for example, querying a very large database, that could only be effectively performed with several large and powerful computers working together as a server. Using the Client / Server model, smaller and less powerful "client" computers can connect to the powerful server computer and request query results. Our Raspberry Pi or phones are examples of "less powerful" computers. With a Client / Server connection, we can greatly magnify the usefulness of our smaller computers by asking for help from a larger server.